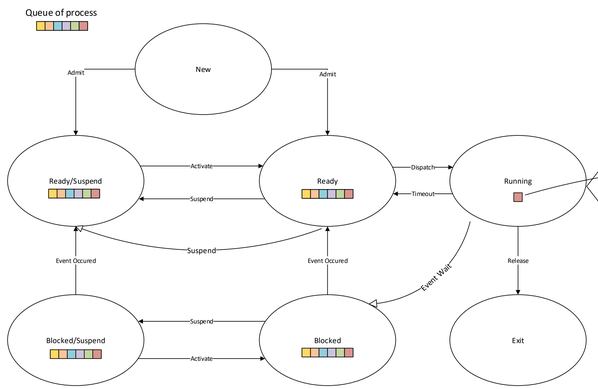
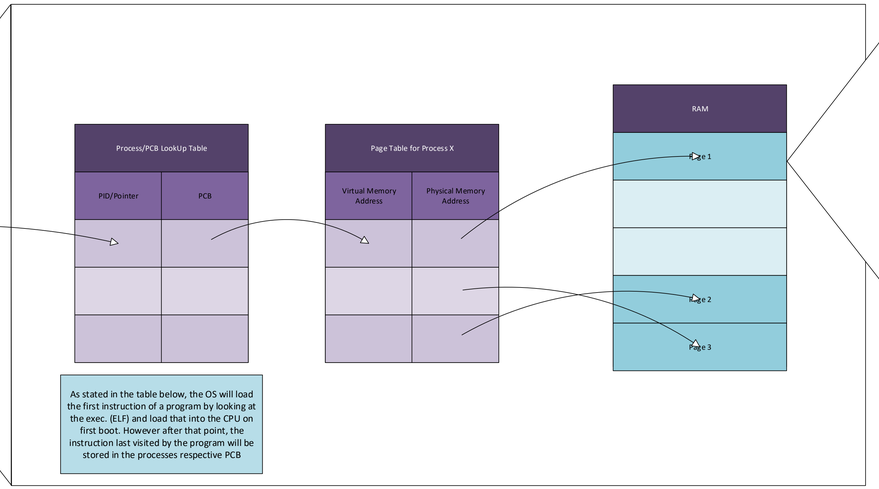
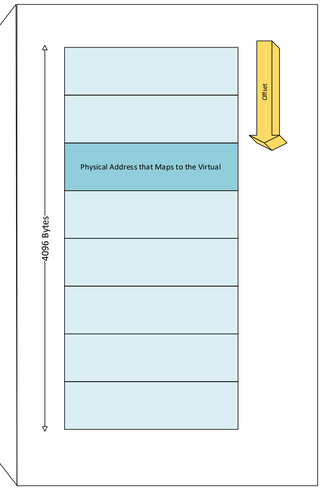




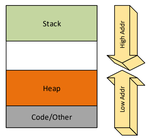


Starting a program

When you run a program, the OS looks at the ELF file, in which the header contains the starting virtual memory.

The OS then loads the local virtual memory into the CPUs PC, and then let's the CPU begin executing (i.e. You can continue with the next tab).

Recall that the CPU **is only aware of the virtual memory** it thinks it's in, the concept of stack and heap only exist within virtual memory. In reality, instructions/stored variables/functions can be found anywhere within RAM, however for the sake of simplicity (on the CPU/Program side), it's assumed that the stack is in "high" memory locations, the heap is generally "lower" locations, and that the acutal code of the program is even lower than the heap.



CPU registers for reference:

R0—Function argument 1 / return value
R1—Function argument 2
R2—Function argument 3
R3—Function argument 4
R4—Callee-saved register (local variables)
R5—Callee-saved register
R6—Callee-saved register
R7 :: FP—(sometimes)Frame pointer (optional, ABI-dependent)
R8—Callee-saved register
R9:: SB / TR—Static base / thread register (platform-specific)
R10—Callee—saved register
R11 :: FP—Frame pointer (common in debug builds)
R12 :: IP—Intra-procedure scratch register

R13 :: SP—Stack Pointer (top of current stack)
R14 :: LR— Link Register (return address for function calls)
R15 :: PC—Program Counter (virtual address of next instruction)

Translating CPU Virtual Memory to Physical Memory

Step 1: CPU asks for instruction
CPU sends PC (virtual address) to the MMU.
CPU does not know where this is in physical RAM.

Step 2: MMU translates virtual → physical
MMU looks up the page number in the page table.
MMU adds the offset within the page.
Output: physical RAM address (e.g., frame 17 + offset 0x123).

Step 3: MMU fetches instruction from RAM
Reads instruction bytes from physical RAM.
Returns instruction to CPU.

Step 4: CPU executes the instruction
Performs whatever the instruction says (add, move, branch, etc.).
Updates PC to point to the next virtual address (PC += instruction length or jump target).

Step 5: Repeat
CPU sends next PC to MMU.
MMU translates virtual → physical.
CPU executes.
And so on, billions of times per second.

the MMU does not "look in the PCB" for every instruction. The PCB is only used **during context switches**. After that, the MMU can work independently.

the PCB isn't being read continuously. It's only consulted **to initialize the CPU and MMU when switching processes**.

The PCB is **not consulted again** until the next context switch.